



TITLE:

# Generation of $k$ -permutations in $O(1)$ time per permutation by reversing sublists (Models of Computation and Algorithms)

AUTHOR(S):

Mikawa, Kenji; Semba, Ichiro

---

CITATION:

Mikawa, Kenji ...[et al]. Generation of  $k$ -permutations in  $O(1)$  time per permutation by reversing sublists (Models of Computation and Algorithms). 数理解析研究所講究録 1999, 1093: 5-10

ISSUE DATE:

1999-04

URL:

<http://hdl.handle.net/2433/62981>

RIGHT:

# Generation of $k$ -permutations in $O(1)$ time per permutation by reversing sublists

三河 賢治 (Kenji MIKAWA)      仙波 一郎 (Ichiro SEMBA)

Department of Computer Science, Ibaraki University  
{mikawa, isemba}@cis.ibaraki.ac.jp

## Abstract

We discuss the problem of generating all  $k$ -permutations of  $n$  objects. Several papers have introduced a technique to alternately reverse sublists of a listing for some combinatorial Gray codes in an efficient manner. Our approach is to apply the technique to a listing of all  $k$ -permutations of  $n$  objects constructed recursively by reversing sublists. We show that the list contains  $n!/(n-k)!$  permutations so that each string differs from its predecessor by the transposition of two elements. It is easy to convert the construction to a recursive algorithm and then we develop the algorithm that produces successive permutations in a constant amortized time per permutation.

## 1 Introduction

Many algorithms have been published for generating all permutations of  $n$  objects and then there is a number of listings of successive permutations. One of the listings is the transposition order that is introduced independently by Johnson [3] and Trotter [11]. It is well-known that each permutation differs by the transposition of adjacent elements.

Recently several interesting papers have been achieved for generating some combinatorial Gray codes in a constant or constant amortized time per object [1, 9, 5, 6, 8, 10, 2]. However, it is not trivial to generate a listing of combinatorial Gray codes in a unique manner. Most of those papers managed to give a simple recurrence relation for combinatorial Gray codes. Ruskey generalized a close relationship between some combinatorial Gray codes constructed recursively by reversing sublists [9]. To reverse certain sublists seems to contribute a reduction of differences between successive objects.

A  $k$ -permutation of  $n$  objects is an arrangement of the first  $k$  objects out of  $n$  objects. First, we give a few modified definition for  $k$ -permutations which are extended into  $n$  length strings such that the set of all permutations of  $n$  objects contains the smaller set of the extended  $k$ -permutations. Our approach is to apply the reversing technique to such  $k$ -permutations. Then a listing of all  $k$ -permutations is obtained such that successive strings differ by the transposition of two elements. This paper presents a recursive algorithm for generating them in a constant amortized time per string. It is obtained directly from the recursively defined construction for  $k$ -permutations. Note that we do not count the time for printing permutations.

## 2 Definitions and properties

To begin with, we extend  $k$ -permutations of  $n$  distinct objects to strings of length  $n$ : a  $k$ -permutation of length  $n$  consists of  $n$  elements which the first  $k$  elements are arranged in its original order and the rests are arranged in a lexical order. For example, if a string 52 is a 2-permutation of the set  $\{1, 2, 3, 4, 5\}$ , then its extension is 52134. When it will not lead to confusion, we call them simply  $k$ -permutations.

The following useful notations are defined in [9]. If  $L$  is a list of strings and  $x$  is a symbol, then  $x \cdot L$  denotes the list of strings obtained by appending an  $x$  to each string of  $L$ . For example, if  $L = 12, 21$ , then  $3 \cdot L = 312, 321$ . If  $L$  and  $L'$  are lists then  $L \circ L'$  denotes the concatenation of the two lists. For example, if  $L = 12, 21$  and  $L' = 34, 43$ , then  $L \circ L' = 12, 21, 34, 43$ .

For a list  $L$ , let  $first(L)$  denote the first element on the list and let  $last(L)$  denote the last element on the list. If  $L$  is a list  $l_1, l_2, \dots, l_n$ , then  $\bar{L}$  denotes the list obtained by listing the elements of  $L$  in reverse order; i.e.,  $\bar{L} = l_n, \dots, l_2, l_1$ . Note the obvious equations  $first(\bar{L}) = last(L)$  and  $last(\bar{L}) = first(L)$ .

Let  $T_k(n)$  be a listing of all  $k$ -permutations of the set  $\{p_1, p_2, \dots, p_n\}$ . The construction for the lists consists of two parts, one of which generates  $k$  length permutations in their original order and the other of which generates  $n - k$  length permutations in a lexical order. The following construction is the case for the original part. The list involves  $n$  recursively defined sublists which are alternately reversed.

$$T_k(n) = \begin{cases} \pi_1 \cdot T_{k-1}(n-1) \circ \pi_2 \cdot \overline{T_{k-1}(n-1)} \circ \dots \\ \quad \dots \circ \pi_{n-1} \cdot \overline{T_{k-1}(n-1)} \circ \pi_n \cdot T_{k-1}(n-1) & \text{if odd } n, \\ \pi_1 \cdot T_{k-1}(n-1) \circ \pi_2 \cdot \overline{T_{k-1}(n-1)} \circ \dots \\ \quad \dots \circ \pi_{n-1} \cdot T_{k-1}(n-1) \circ \pi_n \cdot \overline{T_{k-1}(n-1)} & \text{if even } n, \end{cases}$$

and the case for the lexical part,

$$\mathbf{T}_k(n) = \pi_1 \cdot \mathbf{T}_{k-1}(n-1).$$

These are subject to the terminal condition that  $\mathbf{T}_k(0) = \emptyset$ . The construction appends  $\pi_i$ 's  $\in \{p_1, p_2, \dots, p_n\}$  to sublists in a lexical order from left to right and each sublist is reconstructed with the set obtained by deleting a given element and renumbering the rests from  $\pi_1$  to  $\pi_{n-1}$ . This constraint requires that permutations contain all distinct elements.

LEMMA 2.1 *The list  $\mathbf{T}_k(n)$  satisfies the following properties:*

- (1) *Successive  $k$ -permutations in  $\mathbf{T}_k(n)$  differ in exactly two elements.*
- (2)  *$\text{first}(\mathbf{T}_k(n)) = p_1 p_2 \cdots p_n$ .*
- (3)  *$\text{last}(\mathbf{T}_k(n)) = \begin{cases} p_n p_{n-1} p_1 p_2 \cdots p_{n-2} & \text{if odd } n \text{ and } k \geq 2, \\ p_n p_1 p_2 \cdots p_{n-1} & \text{otherwise.} \end{cases}$*

*Proof.* The proof is by induction on  $n$ . The list obviously has the stated properties for  $1 \leq k \leq n \leq 2$ . Suppose that the lemma is true for  $n \geq 3$ . We must show it to be correct for  $n+1$ . For convenience, we assume the  $i$ th element in a permutation to be placed in the position  $n-i$ , that is, the last element is placed in the position 0.

Obviously the list  $\mathbf{T}_1(n+1)$  contains  $n+1$  permutations in which the  $i$ th permutation is  $p_i p_1 \cdots p_{i-1} p_{i+1} \cdots p_{n+1}$  and the permutation differs from its predecessor by two elements in positions  $n$  and  $n-i$ . Otherwise, for  $k \geq 2$ , the list contains  $n+1$  sublists and we need to inspect the transposition of successive permutations at the interface between the  $i$ th sublist and the  $(i+1)$ st sublist. The transposition behaves in different ways according to the parities  $n$  and  $i$ .

The first case is for even  $i$ . The  $i$ th sublist is reverse and the  $(i+1)$ st sublist is natural. The contiguous permutations between the  $i$ th sublist and the  $(i+1)$ st sublist differ by two elements, since the last permutation of the  $i$ th sublist is the lexically first one, as shown below.

$$\begin{array}{l} p_i \cdot \overline{\mathbf{T}_{k-1}(n)} \left\{ \begin{array}{l} \vdots \\ \underline{p_i} \ p_1 \cdots p_{i-1} \ \underline{p_{i+1}} \ p_{i+2} \cdots p_{n+1} \end{array} \right. \\ p_{i+1} \cdot \mathbf{T}_{k-1}(n) \left\{ \begin{array}{l} p_{i+1} \ p_1 \cdots p_{i-1} \ p_i \ p_{i+2} \cdots p_{n+1} \\ \vdots \end{array} \right. \end{array}$$

The underlined elements that are swapped appear in positions  $n$  and  $n-i$ . When odd  $n+1$ , this case occurs on the last interface. The third property

holds, since the last permutation of  $\mathbf{T}_k(n+1)$  is  $p_{n+1} \cdot \text{last}(\mathbf{T}_{k-1}(n))$ , that is,  $p_{n+1} p_n p_1 \cdots p_{n-1}$ .

The second case is for odd  $i$ . The  $i$ th sublist is natural and the  $(i+1)$ st sublist is reverse. (1) When odd  $n+1$ , the contiguous permutations between the  $i$ th sublist and the  $(i+1)$ st sublist are shown below.

$$\begin{aligned} p_i \cdot \mathbf{T}_{k-1}(n) &= \begin{cases} p_i & p_1 \cdots p_{i-1} p_{i+1} \cdots p_{n+1} \\ \vdots & \\ \underline{p_i} & p_{n+1} p_1 \cdots p_{i-1} \underline{p_{i+1}} & p_{i+2} \cdots p_n \end{cases} \\ p_{i+1} \cdot \overline{\mathbf{T}_{k-1}(n)} &= \begin{cases} p_{i+1} & p_{n+1} p_1 \cdots p_{i-1} & p_i & p_{i+2} \cdots p_n \\ \vdots & \\ p_{i+1} & p_1 \cdots p_i p_{i+2} \cdots p_{n+1} \end{cases} \end{aligned}$$

The underlined elements that are swapped appear in positions  $n$  and  $n-i-1$ . (2) When even  $n+1$ , we can give some formulae for detecting the transposition of successive permutations at each interface. The successive permutations at the last interface differ by the elements in positions  $n$  and  $n-1$ , as shown below.

$$\begin{aligned} p_n \cdot \mathbf{T}_{k-1}(n) &= \begin{cases} p_n & p_1 \cdots p_{n-1} p_{n+1} \\ \vdots & \\ \underline{p_n} & \underline{p_{n+1}} & p_1 \cdots p_{n-1} \end{cases} \\ p_{n+1} \cdot \overline{\mathbf{T}_{k-1}(n)} &= \begin{cases} p_{n+1} & p_n & p_1 \cdots p_{n-1} \\ \vdots & \\ p_{n+1} & p_1 \cdots p_{n-1} p_n \end{cases} \end{aligned}$$

The last property holds in this case. Otherwise, for  $i < n$ , the transposition behaves in two different ways depending upon the value of  $k$ . When  $k=2$ , the elements of permutations that appear in positions greater than 2 are arranged in a lexical order. The contiguous permutations between the  $i$ th sublist and the  $(i+1)$ st sublist are identical in arrangements as the ones for the case (1), that is, the elements that are swapped appear in positions  $n$  and  $n-i-1$ . When  $k > 2$ , they are shown below.

$$\begin{aligned} p_i \cdot \mathbf{T}_{k-1}(n) &= \begin{cases} p_i & p_1 \cdots p_{i-1} p_{i+1} \cdots p_{n+1} \\ \vdots & \\ \underline{p_i} & p_{n+1} p_n p_1 \cdots p_{i-1} \underline{p_{i+1}} & p_{i+2} \cdots p_{n-1} \end{cases} \\ p_{i+1} \cdot \overline{\mathbf{T}_{k-1}(n)} &= \begin{cases} p_{i+1} & p_{n+1} p_n p_1 \cdots p_{i-1} & p_i & p_{i+2} \cdots p_{n-1} \\ \vdots & \\ p_{i+1} & p_1 \cdots p_i p_{i+2} \cdots p_{n+1} \end{cases} \end{aligned}$$

The elements that are swapped appear in positions  $n$  and  $n-i-2$ . The list  $\mathbf{T}_k(n+1)$  has the stated properties. The proof is complete. ■

```

procedure interchange(n,k,i:integer);
begin
  if (k=1) or not(odd(i)) then swap(n-1,n-i-1)
  else if odd(n) then swap(n-1,n-i-2)
    else if i=n-1 then swap(n-1,n-2)
      else if k=2 then swap(n-1,n-i-2) else swap(n-1,n-i-3);
end {of procedure};

```

Figure 1: The procedure `interchange(n,k,i)`.

```

procedure gen(n,k:integer);
var i:integer;
begin
  if k>0 then
    for i:=1 to n do begin
      if odd(i) then gen(n-1,k-1) else neg(n-1,k-1);
      if i<n then interchange(n,k,i);
    end
  end {of procedure};

```

Figure 2: The recursive procedure `gen(n,k)`.

### 3 Implementation and analysis

To begin with, we summarize the transposition of successive permutations between the  $i$ th sublist and the  $(i + 1)$ st sublist and show it in a Pascal procedure, in Figure 1. The procedure `swap(i,j)` swaps the elements in positions  $i$  and  $j$ . The definition of the list  $T_k(n)$  leads directly to a recursive algorithm for generating all  $k$ -permutations of  $n$  objects. The Pascal procedure `gen(n,k)` generates the list  $T_k(n)$ , shown in Figure 2 and the procedure `neg(n,k)` is a symmetric procedure of `gen(n,k)` which generates the reversed list  $\overline{T_k(n)}$ .

Let us analyze the running time of `gen(n,k)`. The procedure `gen` does  $n$  recursive calls to either `gen` or `neg` in the while statement. We also know that it calls `interchange` once per loop and the interchange operation takes a constant time to find the two elements that are swapped. Thus the total amount of computations is proportional to the number of recursive calls, which is  $O(n! / (n - k)!)$ . To summarize above, the procedure `gen` generates all  $k$ -permutations in an amortized constant time to go from one string to the next.

## References

- [1] J. R. Bitner, G. Ehrlich, and E. M. Reingold, Efficient generation of the binary reflected Gray code and its applications, *Comm. Assoc. Comput. Mach.* **19** (1976), 517–521.
- [2] B. Bultena and F. Ruskey, An Eades-McKay algorithm for well-formed parentheses strings, to appear in *Inform. Proc. Lett.*
- [3] S. M. Johnson, Generation of permutations by adjacent transpositions, *Math. Comp.* **17** (1963), 282–285.
- [4] Y. Koda and F. Ruskey, A gray code for the ideals of a forest poset, *J. Algorithms* **15** (1993), 324–340.
- [5] J. Lucas, The rotation graph of binary tree is Hamiltonian, *J. Algorithms* **8** (1987), 503–535.
- [6] J. Lucas, D. Roelants van Baronaigien, and F. Ruskey, On rotations and the generation of binary trees, *J. Algorithms* **15** (1993), 343–366.
- [7] K. Mikawa and T. Takaoka, Generation of balanced parenthesis strings in  $O(1)$  time per string, submitted to *J. Algorithms*.
- [8] A. Proskurowski and F. Ruskey, Binary tree Gray codes, *J. Algorithms* **6** (1985), 225–238.
- [9] F. Ruskey, Simple combinatorial Gray codes constructed by revering sublists, *4th ISAAC, Lecture Notes in Comput. Sci., Springer-Verlag*, **761** (1993), 201–208.
- [10] F. Ruskey and A. Proskurowski, Generating binary trees by transpositions, *J. Algorithms* **11** (1990), 68–84.
- [11] H. F. Trotter, Algorithm 115: Perm, *Comm. Assoc. Comput. Mach.* **5** (1962), 434–435.